

# TOPICS

- User Space/Kernel Space
- System Calls
- Basics of Pointer

# User Space Vs Kernel Space

- **Kernel space** is where the kernel (i.e., the core of the operating system) runs and provides its *services*. It's something that the user is not allowed to interfere with.
- **User Space** is that portion of system memory in which *user processes* runs.

## User Space vs Kernel Space

### User Space

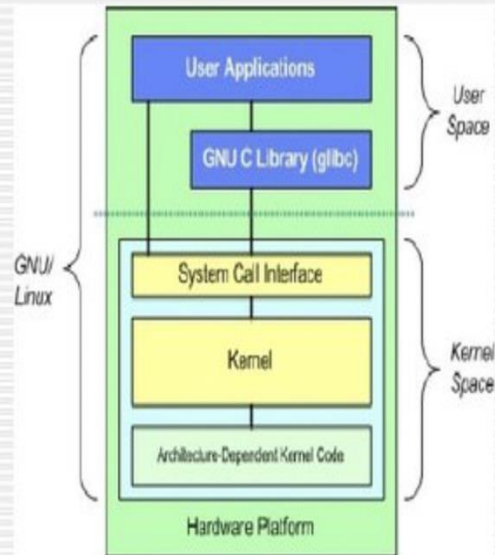
▣ User space is the memory area where all user mode application work and this memory can be swapped out when necessary

▣ User space process normally runs in its own virtual memory space and unless explicitly requested, cannot access the memory of other processes.

### Kernel Space

▣ Kernel Space is strictly reserved for running the kernel (OS background process), kernel extensions and most device drivers

▣ Linux kernel space gives full access to the hardware, although some exceptions runs in user space. (The graphic system most people use with Linux does not run in kernel in contrast to that found in Microsoft Windows)



# System Calls

- A programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
- Each system call corresponds to a number defined in a syscalls table.
- **syscall(number, ...)**

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0      common  read      sys_read
1      common  write     sys_write
2      common  open      sys_open
3      common  close     sys_close
4      common  stat      sys_newstat
5      common  fstat     sys_newfstat
6      common  lstat     sys_newlstat
7      common  poll      sys_poll
8      common  lseek     sys_lseek
9      common  mmap      sys_mmap
10     common  mprotect  sys_mprotect
11     common  munmap    sys_munmap
12     common  brk       sys_brk
13     64      rt_sigaction sys_rt_sigaction
14     common  rt_sigprocmask sys_rt_sigprocmask
--More-- (4%)
```

# Types of system calls:

There are 5 different categories of system calls

1. Process control: end, abort, create, terminate, allocate and free memory.
2. File management: create, open, close, delete, read file etc.
3. Device management
4. Information maintenance
5. Communication

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()

Information	GetCurrentProcessID()	getpid()
Maintenance	SetTimer() Sleep()	alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

# Do we use system calls at all?

- **Example 1:**

- If you open a file using `fopen()` in the library `stdio.h`, it gets translated into the **`open()`** system call.
- In the standard library, the user-space implementation of the `open()` system call executes and passes the system call number.

- **Example 2:**

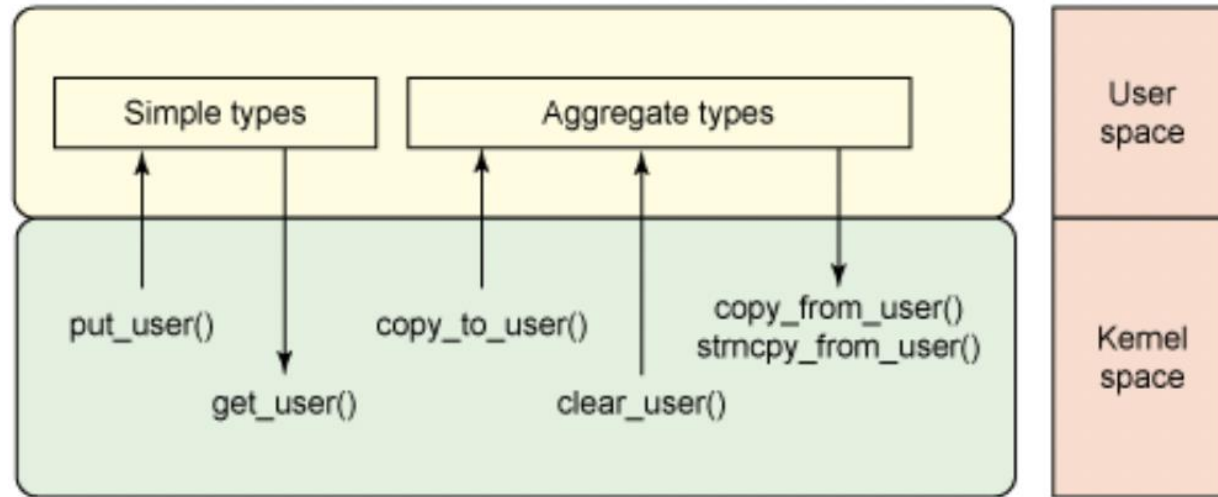
- In Unix-like systems, `fork()` and `exec()` are C- library functions that in turn execute instructions that invoke the **`fork()`** and **`exec()`** system calls.

# Executing a System Call

1. A user space program invokes the syscall
2. A (typically) software interrupt called a trap is triggered (INT)
3. Mode bit is flipped from user to kernel (1 to 0).
4. The interrupt tells the kernel which syscall was called.
  - a. Requisite data may be passed in
  - b. The kernel verifies if all parameters are legal before executing the system call
5. After execution, mode bit flips and user program resumes



# Data Movement Between user space and kernel Space



# The User Space Memory Access API

Function	Description
<code>access_ok</code>	Checks the validity of the user space memory pointer
<code>get_user</code>	Gets a simple variable from user space
<code>put_user</code>	Puts a simple variable to user space
<code>clear_user</code>	Clears, or zeros, a block in user space
<code>copy_to_user</code>	Copies a block of data from the kernel to user space
<code>copy_from_user</code>	Copies a block of data from user space to the kernel
<code>strnlen_user</code>	Gets the size of a string buffer in user space
<code>strncpy_from_user</code>	Copies a string from user space into the kernel

# *Why can't you just call, say, memcpy?*

Two reasons:

1. The kernel is capable of writing to any memory. User process's can't.  
**copy\_to\_user** needs to check **dst** to ensure it is accessible and writable by the current process.
2. Depending on the architecture, you can't simply copy data from kernel to user-space. You might need to do some special setup first, invalidate certain caches, or use special operations.

# Pointers

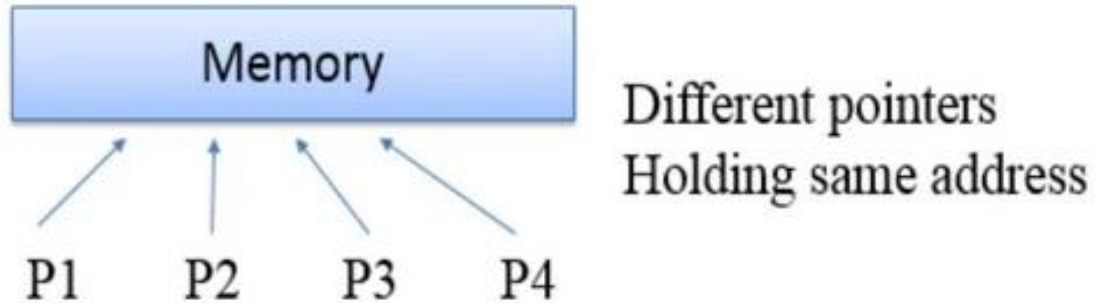
- A variable that holds a memory address
- This address is the location of other object

Exception:

Null pointer : Doesn't contain any address and it's value is 0

Pointer can have 3 types of content associated with:

1. Address
2. Null
3. Invalid values (Doesn't correspond to any memory location)



**If you modify/delete the memory location, same is reflected in all pointers**

# Declaring pointer

- Data-type \* name
- \* is a unary operator, called dereference operator
- \* is used to declare the pointer and also to dereference it(Finding the value at that particular memory location)

```
int * ptr;
```

The variable “ptr” stores a pointer to an “int”

# Pointer operations in c

- Creation : Returns variable's memory address
- Dereference : Returns contents stored at address
- Indirect Assignment : Stores value at address
  - `*pointer = value`
- Direct Assignment : Stores pointer in another variable
  - `Pointer1 = pointer2`



## Simple c program:

```
int x;
```

```
int *p;
```

```
x = 0;
```

```
p = &x;
```

```
*p = 1;
```

What will be the value of x?

# Exercise 1:

```
Int i1 = 1;
```

```
Int i2 = 2;
```

```
Int *ptr1;
```

```
Int *ptr2;
```

```
Ptr1 = &i1;
```

```
Ptr2 = ptr1;
```

```
*ptr1 = 3
```

```
i2 = *ptr2;
```

## Exercise 2:

Int i = 100;

Int j = 200;

Int \*ptr1 = &i;

Int \*ptr2 = &j;

Ptr1 = \*ptr2;

Valid?

Ptr1 = ptr2;

What's the value of i and j?

# Pass by reference

```
Void setvalues(int *x, int *y) {
```

```
    *x = 100;
```

```
    *y = 200;
```

```
}
```

```
Void callfunction(void) {
```

```
    x = 1;
```

```
    y = 2;
```

```
    setvalues(&x, &y);
```

```
}
```

## Extra topics for fun:

- Pointer for strings
- Pointer Arithmetic
- Pointer to Pointer



**Thank You!**